Complexité des algorithmes

En théorie, construire un algorithme permettant de résoudre un problème peut être suffisant, par exemple pour déterminer l'existence de certains éléments en mathématiques. En pratique, il faut que les algorithmes soient utilisables en pratique ; un algorithme qui demande trop de temps ou d'espace mémoire pour calculer son résultat sera inutile.

On introduit alors les concepts de complexité ; on va essayer d'estimer le nombre d'opérations que fait un algorithme pour calculer son résultat. Il suffira alors de multiplier ce nombre par le temps d'exécution d'une seule opération (temps qui dépend de l'opération de base choisie, et de la machine sur laquelle on lance l'algorithme) pour estimer le temps total d'exécution de l'algorithme.

On ne s'intéressera qu'à la complexité temporelle ; depuis quelques années, l'espace mémoire disponible est gigantesque, et il est plus rare d'essayer de limiter l'utilisation de cet espace.

1 Les différentes classes de complexité

MÉTHODE :

Pour calculer une complexité :

-
-
-

EXEMPLE

Regardons l'algorithme de calcul de la taille d'une liste :

```
let taille li =
  match n with
    |[] -> 0
    |t::q -> 1+(taille q)
```

Pour cet algorithme, on va calculer la complexité en terme de
. La taille de l'entrée sera

Soit donc C_n le nombre d'additions effectuées pour une liste de n éléments. On a donc

On a donc une suite arithmétique, et pour tout n,

En général, on ne pourra pas calculer exactement le nombre d'opérations faite ; on se contentera des les estimer.

Définition 1.1
Soient (u_n) et (v_n) deux suites. On dit que

- (u_n) *et* (v_n) *sont* équivalentes *si*
- u *est* dominée *par* v *si*
- u *est un* théta *de* v *s'il existe* $A, B > 0$ *tels que*

En général, on exprimera les complexités avec des grands "O". Attention, cette notation est peu précise ; on sait seulement qu'une certaine suite est majorée, mais cette borne peut être immense.

EXEMPLE

On a $10^{100}n =$, mais un algorithme demandant $10^{100}n$ opérations sera inutilisable.

Définition 1.2
On donne des noms à certaines classes de complexité, données dans le tableau suivant :

Nom de la classe	*Complexité*

En pratique, on essaye d'utiliser des algorithmes au plus quadratiques, voire polynomiaux. À partir de la complexité exponentielle, on ne peut pas utiliser l'algorithme, sauf pour des entrées de très petite taille.

2 Calculs de complexité

Il y a certaines règles à savoir utiliser pour calculer des complexités, qui sont différentes selon que l'algorithme est en style impératif ou récursif.

2.1 Programmation impérative

Les différentes opérations à considérer sont les suites d'instructions, les conditionnelles et les boucles.

On notera C_P le nombre d'opérations fait pour exécuter le bloc P d'instructions.

Proposition 2.1
Si P et Q sont deux séquences d'instructions, alors

Proposition 2.2
Si P et Q sont deux séquences d'instructions et b un booléen, alors

Proposition 2.3
Si P_i sont des séquences d'instructions, alors

EXEMPLE

Regardons le code de la factorielle :

```
let fact i =
  let f = ref 1 in
  for i = 1 to n do
    f := f*i
  done;
  !f;;
```

Alors, l'instruction "f :=f*i" faisant une multiplication, le programme fera multiplications.

Si on considère le changement de la valeur d'une référence comme une opération élémentaire, on en fera aussi.

Pour les boucles conditionnelles du type "while b do P done", on essaye de majorer le nombre d'itérations qu'on va faire, et on multiplie ce nombre par T_P.

2.2 Programmation récursive

Dans le cas d'algorithme récursif, on trouve très souvent des relations de récurrences vérifiées par la complexité.

EXEMPLE

Regardons le calcul de la factorielle

```
let rec fact n =
  if n=0 then 1 else n*fact(n-1)
```

On a alors la relation $C_0 = 0$ et

Il est donc important de savoir résoudre des équations de récurrences.

Commençons par les relations du type

$$C_{n+1} = aC_n + f(n)$$

où $a \in \mathbb{N}^*$ et $f : \mathbb{N} \to \mathbb{N}$.

Proposition 2.4
Soient donc a, f, C définies comme précédemment. Alors

- *si $a = 1$, alors pour tout n,*

$$C_n =$$

- *si $a \geqslant 2$, alors pour tout n,*

$$\frac{C_n}{a^n} =$$

Démonstration. Dans le cas où $a = 1$, une récurrence immédiate permet de conclure. Dans le cas $a \geqslant 2$, il suffit de faire le changement de variable $u_n = \frac{C_n}{a^n}$, et on est ramenés au cas précédent. □

Pour ce type de récurrence, on a trois cas pour la complexité asymptotique :

- si la série $\sum \frac{f(k)}{a^k}$ converge, alors
- si $f(n) = \mathrm{O}(a^n)$, alors
- si $f(n) = \mathrm{O}(b^n)$, $b > a$, alors

EXEMPLE

Soit la fonction suivante :

```
let rec minim l =
  match l with
    |[] -> failwith "minim"
    |[x] -> x
    |t::q -> if x< minim q then x else minim q
```

On compte le nombre de comparaisons. On a donc $C_0 = C_1 = 0$, et

On a donc, avec les notations précédentes, . La série converge, donc

Dans le cas d'algorithmes de type "diviser pour régner", les relations de récurrences obtenues seront plutôt du type

$$C_n = aC_{\frac{n}{2}} + d_n$$

où d_n est le coût du partage et de la fusion.

On a alors le *master theorem* :

Théorème 2.5 : master
On suppose que $d_n = \mathrm{O}\left(n^d\right)$, $d \in \mathbb{R}$. *Alors*

- *si* $d < \lg(a)$, *alors*
- *si* $d = \lg(a)$, *alors*
- *si* $d > \lg(a)$, *alors*

EXEMPLE

Considérons l'algorithme d'exponentiation rapide :

```
let rec quickexp x n =
  match n with
    |0 -> 1
    |_ when n mod 2 = 0 -> quickexp (x*x) (n/2)
```

```
    |_ -> x*(quickexp (x*x) ((n-1)/2)
```

Regardons la complexité dans le cas $n = 2^p$. On a alors

Ici, on a $f_n =$, donc $f_n =$. On est alors dans le second cas du théorème maître, et donc

Dans le cas où n n'est pas une puissance de 2, on l'encadre entre deux puissances de 2, et on retrouve

3 Exercices

Exercice 1. Pour le calcul de la suite de Fibonacci, calculer la complexité de chacun des algorithmes donnés dans le premier TP.

Exercice 2. Dans le TP sur les listes, donner la complexité des fonctions implémentées.

Exercice 3. Dans le TP sur les tableaux, donner la complexité des fonctions calculant le maximum, vérifiant si un tableau est un palindrome, et du produit de matrices.

Exercice 4. Calculer les complexités des différents algorithmes de tri, sauf du tri rapide.

Exercice 5. L'objectif de cet exercice est de calculer la complexité du tri rapide. On supposera qu'on a n entiers à trier, qui sont tous les entiers de $[\![1, n]\!]$. On note $C(n)$ la complexité de l'algorithme en terme de nombre de comparaisons quand la liste est à n éléments.

On considère que l'opération de fusion se fait en temps constant.

a) Estimer la complexité dans le pire cas du tri rapide.

b) On s'intéresse maintenant à la complexité moyenne. On suppose que chaque entier à une probabilité $\frac{1}{n}$ d'être le pivot (autrement dit, on considère que la liste est mélangée aléatoirement à chaque fois).

Montrer que si $C_k(n)$ est la complexité moyenne quand le pivot est k, la complexité moyenne vérifie

$$C(n) = \frac{1}{n} \sum_{k=1}^{n} C_k(n).$$

c) Montrer que

$$C_k(n) = n + 1 + C(k-1) + C(n-k).$$

d) En déduire la valeur de $C(n)$ en fonctions des $C(k)$.

e) On pose $D(n) = nC(n)$. Montrer que

$$D(n+1) - D(n) = 2(n+1) + 2C(n).$$

f) En déduire que

$$\frac{C(n+1)}{n+2} - \frac{C(n)}{n+1} = \frac{2}{n+2'},$$

puis la valeur de $C(n)$.

g) En déduire que $C(n) = \mathrm{O}(n \log n)$.